A Sound Polymorphic Type System for a Dialect of C[⋆]

Geoffrey Smith

School of Computer Science, Florida International University, Miami, FL 33199, USA, email: smithg@cs.fiu.edu

Dennis Volpano

Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA, email: volpano@cs.nps.navy.mil

Advanced polymorphic type systems have come to play an important role in the world of functional programming. But, so far, these type systems have had little impact upon widely-used imperative programming languages like C and C++. We show that ML-style polymorphism can be integrated smoothly into a dialect of C, which we call *Polymorphic C.* It has the same pointer operations as C, including the address-of operator `&`, the dereferencing operator `*`, and pointer arithmetic. We give a natural semantics for Polymorphic C, and prove a type soundness theorem that gives a rigorous and useful characterization of what can go wrong when a well-typed Polymorphic C program is executed. For example, a well-typed Polymorphic C program may fail to terminate, or it may abort due to a dangling pointer error. Proving such a type soundness theorem requires a notion of an *attempted program execution*; we show that a natural semantics gives rise quite naturally to a transition semantics, which we call a *natural transition semantics*, that models program execution in terms of transformations of *partial derivation trees*. This technique should be generally useful in proving type soundness theorems for languages defined using natural semantics.

1 Introduction

Much attention has been given to developing sound polymorphic type systems for languages with imperative features. Most notable is the large body of

[⋆] To appear in *Science of Computer Programming*, 1998. This material is based upon work supported by the National Science Foundation under Grants No. CCR-9414421 and CCR-9400592.

Preprint submitted to Elsevier Preprint *31 March 1997*

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **01 MAR 1997**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **A Sound Polymorphic Type System for a Dialect of C**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **School of Computer Science Florida International University Miami, FL 33199**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**28**	

work surrounding ML [4,21,14,5,11,24,22,19]. However, none of these efforts addresses the polymorphic typing of a language that combines variables, arrays and pointers (first-class references), which are key ingredients of traditional imperative languages. As a result, they cannot be directly applied to get ML-style polymorphic extensions of widely-used languages like C and C++.

This paper presents a provably-sound type system for a polymorphic dialect of C, called Polymorphic C. It has the same pointer operations as C, including the address-of operator &, the dereferencing operator *, and pointer arithmetic. The type system allows these operations without any restrictions on them so that programmers can enjoy C's pointer flexibility and yet have type security and polymorphism as in ML. Also, although we do not address it here, it is straightforward to do type inference for Polymorphic C, so that programs need not be written with type annotations [16]. Our type system thus demonstrates that ML-style polymorphism can be brought cleanly into the realm of traditional imperative languages.

We establish the soundness of our type system with respect to a natural semantics for Polymorphic C. First we use Harper's syntactic approach [8] to establish the *type preservation* property (also known as the *subject reduction* property). We then prove a type soundness theorem that gives a rigorous and useful characterization of what can go wrong when a well-typed Polymorphic C program is executed. More precisely, we show that the execution of a well-typed Polymorphic C program either succeeds, fails to terminate, or aborts due to one of a specific set of errors, such as an attempt to dereference a dangling pointer. Proving such a type soundness theorem requires a notion of an *attempted program execution*; we show that a natural semantics gives rise quite naturally to a transition semantics, which we call a *natural transition semantics*, that models program execution in terms of transformations of *partial derivation trees*. This technique should be generally useful in proving type soundness theorems for languages defined using natural semantics.

We begin with an overview of Polymorphic C in the next section. Next, Section 3 formally defines its syntax, type system, and semantics. Then, in Sections 4 and 5, we prove the soundness of the type system. We conclude with some discussion.

2 An Overview of Polymorphic C

Polymorphic C is intended to be as close to the core of Kernighan and Ritchie C [12] as possible. In particular, it is stack-based with variables, pointers, and arrays. Pointers are dereferenced explicitly using *, while variables are dereferenced implicitly. Furthermore, pointers are first-class values, but variables are not. Polymorphic C has the same pointer operations as C. A well-typed Polymorphic C program may still suffer from dangling reference and illegal address errors—our focus has not been on eliminating such pointer insecurities, which would require weakening C's expressive power, but rather on adding ML-style polymorphism to C, so that programmers can write polymorphic functions naturally and soundly as they would in ML, rather than by parameterizing functions on data sizes or by casting to pointers of type `void *`.

2.1 An Example

In this paper, we adopt a concrete syntax for Polymorphic C that resembles the syntax of C.[1] For example, here are three Polymorphic C functions:

```
swap(x,y)
{
    var t = *x;

    *x = *y;
    *y = t
}

reverse(a,n)
{
    var i = 0;

    while (i < n-1-i) {
        swap(a+i, a+n-1-i);
        i = i+1
    }
}

swapsections(a,i,n)
{
    reverse(a,i);
    reverse(a+i,n-i);
    reverse(a,n)
}
```

Note that, unlike C, Polymorphic C does not include type annotations in declarations. (Also, Polymorphic C differs from C in the treatment of semicolons.) Function `reverse(a,n)` reverses the elements of array `a[0:n-1]`, and function `swapsections(a,i,n)` uses `reverse` to swap the array sections `a[0:i-1]` and

[1] See [20] for a alternative ML-like syntax that is somewhat more flexible.

a[i:n-1]. This illustrates that in Polymorphic C, as in C, one can manipulate sections of arrays using pointer arithmetic. The construct var x = e_1; e_2 binds x to a new cell initialized to the value of e_1; the scope of the binding is e_2 and the lifetime of the cell ends after e_2 is evaluated. Variable x is dereferenced implicitly. This is achieved via a typing rule that says that if e has type τ *var*, then it also has type τ.

As in C, the call to swap in reverse could equivalently be written as

```
swap(&a[i], &a[n-1-i])
```

and also as in C, array subscripting is syntactic sugar: e_1[e_2] is equivalent to *(e_1+e_2). Arrays themselves are created by the construct arr x[e_1]; e_2, which binds x to a pointer to an uninitialized array whose size is the value of e_1; the scope of x is e_2, and the lifetime of the array ends after e_2 is evaluated.

The type system of Polymorphic C assigns types of the form τ *var* to variables, and types of the form τ *ptr* to pointers.[2] Functions swap, reverse, and swapsections given above are polymorphic; swap has type

$$\forall\alpha\,.\,\alpha\ \mathit{ptr} \times \alpha\ \mathit{ptr} \rightarrow \alpha,$$

reverse has type

$$\forall\alpha\,.\,\alpha\ \mathit{ptr} \times \mathit{int} \rightarrow \mathit{unit},$$

and swapsections has type

$$\forall\alpha\,.\,\alpha\ \mathit{ptr} \times \mathit{int} \times \mathit{int} \rightarrow \mathit{unit}.$$

Type *unit*, which appears in the types of reverse and swapsections, is a degenerate type containing only the value unit; it serves as the type of constructs, like while loops, that do not produce a useful value. Notice that pointer and array types are unified as in C. Also, variable and pointer types are related by symmetric typing rules for & and *:

if $e : \tau\ \mathit{var}$, then &$e : \tau\ \mathit{ptr}$,

and

if $e : \tau\ \mathit{ptr}$, then *$e : \tau\ \mathit{var}$.

[2] We use *ptr* rather than *ref* to avoid confusion with C++ and ML references.

Note that dereferencing in Polymorphic C differs from dereferencing in Standard ML, where if $e : \tau\ \mathit{ref}$, then $!e : \tau$.

Polymorphic C's types are stratified into three levels. There are the ordinary τ (data types) and σ (type schemes) type levels of Damas and Milner's system [2], and a new level called *phrase types*—the terminology is due to Reynolds [17]—containing σ types and variable types of the form $\tau\ \mathit{var}$. This stratification enforces the "second-class" status of variables: for example, the return type of a function must be a data type, so that one cannot write a function that returns a variable. In contrast, pointer types are included among the data types, making pointers first-class values.

2.2 Achieving Type Soundness in Polymorphic C

Much effort has been spent trying to develop sound polymorphic type systems for imperative extensions of core-ML. Especially well-studied is the problem of typing Standard ML's first-class references [21,14,5,11,24]. The problem is easier in a language with variables but no references, such as Edinburgh LCF ML, but subtle problems still arise [4]. The key problem is that a variable can escape its scope via a lambda abstraction as in

$$\textbf{letvar}\ \mathit{stk} := [\,]\ \textbf{in}\ \lambda v.\, \mathit{stk}\ :=\ v :: \mathit{stk}$$

(This evaluates to a *push* function that pushes values v onto a stack, implemented as a list; here $[\,]$ denotes the empty list and :: denotes *cons*.) In this case, the type system must not allow type variables that occur in the type of *stk* to be generalized, or else the list would not be kept homogeneous. Different mechanisms have been proposed for dealing with this problem [4,22,19]

In the context of Polymorphic C, however, we can adopt an especially simple approach. Because Polymorphic C does not have first-class functions, it is not possible to *compute* a polymorphic value in an interesting way; for example, we cannot write curried functions. For this reason, we suffer essentially no loss of language expressiveness by limiting polymorphism to function declarations.

Limiting polymorphism to function declarations ensures the soundness of polymorphic generalizations, but pointers present new problems for type soundness. If one is not careful in formulating the semantics, then the type preservation property may not hold. For example, if a program is allowed to dereference a pointer to a cell that has been deallocated and then reallocated, then the value obtained may have the wrong type. For this reason, our natural semantics has been designed to catch all pointer errors.

3 A Formal Description of Polymorphic C

The syntax of Polymorphic C is given by the following grammar:

$$
\begin{aligned}
e \ ::= {} & x \mid c \mid \texttt{\&}\,e \mid \texttt{*}\,e \mid e_1\texttt{+}e_2 \mid \\
& e_1\,\texttt{[}e_2\texttt{]} \mid e_1\,\texttt{=}\,e_2 \mid e_1\,\texttt{;}\,e_2 \mid \\
& \texttt{if (}e_1\texttt{)}\ \{e_2\}\ \texttt{else}\ \{e_3\} \mid \\
& \texttt{while (}e_1\texttt{)}\ \{e_2\} \mid \\
& \texttt{var}\ x\ \texttt{=}\ e_1\,\texttt{;}\ e_2 \mid \\
& \texttt{arr}\ x\,\texttt{[}e_1\texttt{];}\ e_2 \mid \\
& x\,\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}\ \{e_1\}\ e_2 \mid \\
& e\,\texttt{(}e_1\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{)}
\end{aligned}
$$

Meta-variable x ranges over identifiers, and c over literals (such as integer literals and `unit`). The expression

$$x\,\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}\ \{e_1\}\ e_2$$

is a function declaration; it declares a function x whose scope is e_2. The `+` operator here denotes only pointer arithmetic. In the full language, `+` would be overloaded to denote integer addition as well.

Like C, Polymorphic C has been designed to ensure that function calls can be implemented on a stack without the use of static links or displays. In C, this property is achieved by the restriction that functions can only be defined at top level. Since Polymorphic C allows function declarations anywhere, we instead impose the restriction that *the free identifiers of any function must be declared at top level.* Roughly speaking, a top-level declaration is one whose scope extends all the way to the end of the program. For example, in the program

```
var n = ...;
arr a[...];
f(x) {...}
f(...)
```

the identifiers declared at top level are `n`, `a`, and `f`. So the only identifiers that can occur free in `f` are `n` and `a`.

A subtle difference between C and Polymorphic C is that the formal parameters of a Polymorphic C function are *constants* rather than local variables. Hence the C function `f(x) {b}` is equivalent to

```
f(x) {var x = x; b}
```

in Polymorphic C. Also, Polymorphic C cannot directly express C's internal static variables. For example, the C declaration

```
f(x) {static int n = 0;  b}
```

must be written in Polymorphic C as

```
var n = 0; f(x) {b}
```

where `n` has been uniquely renamed.

3.1 The Type System of Polymorphic C

The types of Polymorphic C are stratified as follows.

$$\begin{array}{lcll}
\tau & ::= & \alpha \mid \mathit{int} \mid \mathit{unit} \mid \tau\ \mathit{ptr} \mid \tau_1 \times \cdots \times \tau_n \rightarrow \tau & (\mathit{data\ types}) \\
\sigma & ::= & \forall \alpha \,.\, \sigma \mid \tau & (\mathit{type\ schemes}) \\
\rho & ::= & \sigma \mid \tau\ \mathit{var} & (\mathit{phrase\ types})
\end{array}$$

Meta-variable α ranges over *type variables.* Compared to the type system of Standard ML [15], all type variables in Polymorphic C are imperative.

The rules of the type system are given in Figures 1 and 2. It is a deductive proof system used to assign types to expressions. Typing judgments have the form

$$\gamma \vdash e : \rho$$

meaning that expression e has type ρ, assuming that γ prescribes phrase types for the free identifiers of e. More precisely, metavariable γ ranges over *identifier typings*, which are finite functions mapping identifiers to phrase types; $\gamma(x)$ is the phrase type assigned to x by γ and $\gamma[x : \rho]$ is a modified identifier typing that assigns phrase type ρ to x and assigns phrase type $\gamma(x')$ to any identifier x' other than x.

(IDENT) $\gamma \vdash x : \tau$ if $\gamma(x) \geq \tau$

(VAR-ID) $\gamma \vdash x : \tau\ \mathit{var}$ if $\gamma(x) = \tau\ \mathit{var}$

(LIT) $\gamma \vdash c : \mathit{int}$ if c is an integer literal

$\gamma \vdash \texttt{unit} : \mathit{unit}$

(R-VAL)
$$\frac{\gamma \vdash e : \tau\ \mathit{var}}{\gamma \vdash e : \tau}$$

(ADDRESS)
$$\frac{\gamma \vdash e : \tau\ \mathit{var}}{\gamma \vdash \texttt{\&}e : \tau\ \mathit{ptr}}$$

(L-VAL)
$$\frac{\gamma \vdash e : \tau\ \mathit{ptr}}{\gamma \vdash \texttt{*}e : \tau\ \mathit{var}}$$

(ARITH)
$$\frac{\gamma \vdash e_1 : \tau\ \mathit{ptr}, \quad \gamma \vdash e_2 : \mathit{int}}{\gamma \vdash e_1\texttt{+}e_2 : \tau\ \mathit{ptr}}$$

(SUBSCRIPT)
$$\frac{\gamma \vdash e_1 : \tau\ \mathit{ptr}, \quad \gamma \vdash e_2 : \mathit{int}}{\gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau\ \mathit{var}}$$

(ASSIGN)
$$\frac{\gamma \vdash e_1 : \tau\ \mathit{var}, \quad \gamma \vdash e_2 : \tau}{\gamma \vdash e_1\texttt{=}e_2 : \tau}$$

(COMPOSE)
$$\frac{\gamma \vdash e_1 : \tau_1, \quad \gamma \vdash e_2 : \tau_2}{\gamma \vdash e_1\texttt{;}e_2 : \tau_2}$$

(IF)
$$\frac{\gamma \vdash e_1 : \mathit{int}, \quad \gamma \vdash e_2 : \tau, \quad \gamma \vdash e_3 : \tau}{\gamma \vdash \texttt{if (}e_1\texttt{)}\ \{e_2\}\ \texttt{else}\ \{e_3\} : \tau}$$

(WHILE)
$$\frac{\gamma \vdash e_1 : \mathit{int}, \quad \gamma \vdash e_2 : \tau}{\gamma \vdash \texttt{while (}e_1\texttt{)}\ \{e_2\} : \mathit{unit}}$$

(LETVAR)
$$\frac{\gamma \vdash e_1 : \tau_1, \quad \gamma[x : \tau_1\ \mathit{var}] \vdash e_2 : \tau_2}{\gamma \vdash \texttt{var}\ x\ \texttt{=}\ e_1\texttt{;}\ e_2 : \tau_2}$$

Fig. 1. Rules of the Type System (Part 1)

(LETARR)
$$\frac{\gamma \vdash e_1 : \mathit{int}, \quad \gamma[x : \tau_1\ \mathit{ptr}] \vdash e_2 : \tau_2}{\gamma \vdash \texttt{arr}\ x\texttt{[}e_1\texttt{];}\ e_2 : \tau_2}$$

(FUN)
$$\frac{\begin{array}{l}\gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau \\ \gamma[x : \mathit{Close}_\gamma(\tau_1 \times \cdots \times \tau_n \to \tau)] \vdash e' : \tau'\end{array}}{\gamma \vdash x\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)}\ \ \{e\}\ \ e' : \tau'}$$

(FUNCALL)
$$\frac{\begin{array}{l}\gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau, \\ \gamma \vdash e_1 : \tau_1, \\ \ldots \\ \gamma \vdash e_n : \tau_n\end{array}}{\gamma \vdash e\texttt{(}e_1\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{)} : \tau}$$

Fig. 2. Rules of the Type System (Part 2)

The *generalization* of a data type τ relative to γ, written $\mathit{Close}_\gamma(\tau)$, is the type scheme $\forall \bar{\alpha} \,.\, \tau$, where $\bar{\alpha}$ is the set of all type variables occurring free in τ but not in γ. Note the use of *Close* in rule (FUN); this is what allows functions to be given polymorphic types.

We say that τ' is a *generic instance* of $\forall \bar{\alpha} \,.\, \tau$, written $\forall \bar{\alpha} \,.\, \tau \geq \tau'$, if there exists a substitution S with domain $\bar{\alpha}$ such that $S\tau = \tau'$. Note that rule (IDENT) allows an identifier x to be given any type τ that is a generic instance of $\gamma(x)$; this is what allows a polymorphic function to be called with different types of arguments. We extend the definition of $\geq$ to type schemes by saying that $\sigma \geq \sigma'$ if $\sigma \geq \tau$ whenever $\sigma' \geq \tau$. Finally, we say that $\gamma \vdash e : \sigma$ if $\gamma \vdash e : \tau$ whenever $\sigma \geq \tau$.

3.2 The Semantics of Polymorphic C

We now give a natural semantics for Polymorphic C. Before we can do this, we need to extend the language syntax to include some semantic values; these new values are the runtime representations of variables, pointers, and functions:

$$e \ ::= \ (a, 1) \ \mid \ (a, 0) \ \mid \ \lambda x_1, \ldots, x_n .\, e$$

Metavariable a here ranges over *addresses*, which are described below. Expression $(a, 1)$ is a *variable* and expression $(a, 0)$ is a *pointer*. Intuitively, a variable or pointer is represented by an address together with a tag bit, which tells whether it should be implicitly dereferenced or not—thus, variables are implicitly dereferenced and pointers are not. Expression $\lambda x_1, \ldots, x_n .\, e$ is a *lambda* *abstraction* that represents a function with formal parameters $x_1, \ldots, x_n$ and body e.

One might expect that addresses would just be natural numbers, but that would not allow the semantics to detect invalid pointer arithmetic. So instead an address is a pair of natural numbers (i, j) where i is the *segment number* and j is the *offset*. Intuitively, we put each variable or array into its own segment. Thus a simple variable has address $(i, 0)$, and an n-element array has addresses $(i, 0), (i, 1), \ldots, (i, n-1)$. Pointer arithmetic involves only the offset of an address, and dereferencing nonexistent or dangling pointers is detected as a "segmentation fault".

Next we identify the set of *values* v, consisting of literals, pointers, and lambda abstractions:

$$v \ ::= \ c \ \mid \ (a, 0) \ \mid \ \lambda x_1, \ldots, x_n.\, e$$

The result of a successful evaluation is always a value.

Finally, we require the notion of a *memory*. A memory μ is a finite function mapping addresses to values, or to the special results **dead** and **uninit**. These results indicate that the cell with that address has been deallocated or is uninitialized, respectively. We write $\mu(a)$ for the contents of address $a \in dom(\mu)$, and we write $\mu[a := v]$ for the memory that assigns value v to address a, and value $\mu(a')$ to any address a' other than a. Note that $\mu[a := v]$ is an *update* of μ if $a \in dom(\mu)$ and an *extension* of μ if $a \notin dom(\mu)$.

We can now define the *evaluation relation*

$$\mu \vdash e \Rightarrow v, \mu'$$

which asserts that evaluating closed expression e in memory μ results in value v and new memory μ'. The evaluation rules are given in Figures 3 and 4.

We write $[e'/x]e$ to denote the capture-avoiding substitution of e' for all free occurrences of x in e. Note the use of substitution in rules (BINDVAR), (BINDARR), (BINDFUN), and (APPLY). It allows us to avoid environments and closures in the semantics, so that the result of evaluating a Polymorphic C expression is just another Polymorphic C expression. This is made possible by the flexible syntax of the language and the fact that only closed expressions are ever evaluated during the evaluation of a closed expression.

We remark that rule (APPLY) specifies that function arguments are evaluated left to right; C leaves the evaluation order unspecified. Also, note that if there were no `&` operator, there would be no need to specify in rule (BINDVAR) that

(VAL) $$\mu \vdash v \Rightarrow v, \mu$$

(CONTENTS) $$\frac{a \in \mathit{dom}(\mu) \text{ and } \mu(a) \text{ is a value}}{\mu \vdash (a,1) \Rightarrow \mu(a), \mu}$$

(DEREF) $$\frac{\begin{array}{l}\mu \vdash e \Rightarrow (a,0), \mu' \\ a \in \mathit{dom}(\mu') \text{ and } \mu'(a) \text{ is a value}\end{array}}{\mu \vdash \texttt{*}e \Rightarrow \mu'(a), \mu'}$$

(REF) $$\mu \vdash \texttt{\&}(a,1) \Rightarrow (a,0), \mu$$

$$\frac{\mu \vdash e \Rightarrow (a,0), \mu'}{\mu \vdash \texttt{\&*}e \Rightarrow (a,0), \mu'}$$

(OFFSET) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow ((i,j),0), \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow n, \mu' \ \ (n \text{ an integer})\end{array}}{\mu \vdash e_1\texttt{+}e_2 \Rightarrow ((i,j+n),0), \mu'}$$

(UPDATE) $$\frac{\begin{array}{l}\mu \vdash e \Rightarrow v, \mu' \\ a \in \mathit{dom}(\mu') \text{ and } \mu'(a) \neq \textbf{dead}\end{array}}{\mu \vdash (a,1)\texttt{=}e \Rightarrow v, \mu'[a := v]}$$

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow (a,0), \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v, \mu_2 \\ a \in \mathit{dom}(\mu_2) \text{ and } \mu_2(a) \neq \textbf{dead}\end{array}}{\mu \vdash \texttt{*}e_1\texttt{=}e_2 \Rightarrow v, \mu_2[a := v]}$$

(SEQUENCE) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash e_1\,\texttt{;}\,e_2 \Rightarrow v_2, \mu_2}$$

(BRANCH) $$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow n, \mu_1 \ \ (n \text{ a nonzero integer}) \\ \mu_1 \vdash e_2 \Rightarrow v, \mu'\end{array}}{\mu \vdash \texttt{if (}e_1\texttt{)}\ \{e_2\}\ \texttt{else}\ \{e_3\} \Rightarrow v, \mu'}$$

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow \texttt{0}, \mu_1 \\ \mu_1 \vdash e_3 \Rightarrow v, \mu'\end{array}}{\mu \vdash \texttt{if (}e_1\texttt{)}\ \{e_2\}\ \texttt{else}\ \{e_3\} \Rightarrow v, \mu'}$$

Fig. 3. The Evaluation Rules (Part 1)

$$(\textsc{loop})\qquad \frac{\mu \vdash e_1 \Rightarrow \texttt{0}, \mu_1}{\mu \vdash \texttt{while (}e_1\texttt{) \{}e_2\texttt{\}} \Rightarrow \texttt{unit}, \mu_1}$$

$$\frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow n, \mu_1 \;\; (n \text{ a nonzero integer}) \\ \mu_1 \vdash e_2 \Rightarrow v, \mu_2 \\ \mu_2 \vdash \texttt{while (}e_1\texttt{) \{}e_2\texttt{\}} \Rightarrow \texttt{unit}, \mu' \end{array}}{\mu \vdash \texttt{while (}e_1\texttt{) \{}e_2\texttt{\}} \Rightarrow \texttt{unit}, \mu'}$$

$$(\textsc{bindvar})\qquad \frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ (i,0) \notin \mathit{dom}(\mu_1) \\ \mu_1[(i,0) := v_1] \vdash [((i,0),1)/x]e_2 \Rightarrow v_2, \mu_2 \end{array}}{\mu \vdash \texttt{var } x \texttt{ = } e_1\texttt{; } e_2 \Rightarrow v_2, \mu_2[(i,0) := \mathbf{dead}]}$$

$$(\textsc{bindarr})\qquad \frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow n, \mu_1 \;\; (n \text{ a positive integer}) \\ (i,0) \notin \mathit{dom}(\mu_1) \\ \mu_1[(i,0), \ldots, (i,n-1) := \mathbf{uninit}, \ldots, \mathbf{uninit}] \vdash \\ \qquad\qquad [((i,0),0)/x]e_2 \Rightarrow v_2, \mu_2 \end{array}}{\begin{array}{l} \mu \vdash \texttt{arr } x\texttt{[}e_1\texttt{]; } e_2 \Rightarrow \\ \qquad\qquad v_2, \mu_2[(i,0), \ldots, (i,n-1) := \mathbf{dead}, \ldots, \mathbf{dead}] \end{array}}$$

$$(\textsc{bindfun})\qquad \frac{\mu \vdash [\lambda x_1, \ldots, x_n.\, e/x]e' \Rightarrow v, \mu'}{\mu \vdash x\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{) \{}e\texttt{\}}\ e' \Rightarrow v, \mu'}$$

$$(\textsc{apply})\qquad \frac{\begin{array}{l} \mu \vdash e \Rightarrow \lambda x_1, \ldots, x_n.\, e', \mu_1 \\ \mu_1 \vdash e_1 \Rightarrow v_1, \mu_2 \\ \cdots \\ \mu_n \vdash e_n \Rightarrow v_n, \mu_{n+1} \\ \mu_{n+1} \vdash [v_1, \ldots, v_n/x_1, \ldots, x_n]e' \Rightarrow v, \mu' \end{array}}{\mu \vdash e\texttt{(}e_1\texttt{, }\ldots\texttt{ ,}e_n\texttt{)} \Rightarrow v, \mu'}$$

Fig. 4. The Evaluation Rules (Part 2)

a variable dies at the end of its scope; it would simply become unreachable at that point (and its storage could be reused).

Note that a successful evaluation always produces a value and a memory:

Lemma 1 *If $\mu \vdash e \Rightarrow v, \mu'$, then v is a value and μ' is a memory.*

PROOF. By induction on the structure of the derivation. □

4 Type Preservation

We now turn to the question of the soundness of our type system. We begin in this section by using the framework of Harper [8] to prove that our type system satisfies the *type preservation* property (sometimes called the *subject reduction* property). This property basically asserts that types are preserved across evaluations; that is, if an expression of type τ evaluates successfully, it produces a value of type τ. But before we can do this, we need to extend our typing rules so that we can type the semantic values (variables, pointers, and lambda abstractions) introduced in Section 3.2.

Typing a variable $(a, 1)$ or a pointer $(a, 0)$ clearly requires information about the type of value stored at address a; this information is provided by an *address typing* λ. One might expect an address typing to map addresses to data types. This turns out not to work, however, because a well-typed program can produce as its value a nonexistent pointer, and such pointers must therefore be typable if type preservation is to hold. For example, the program

```
arr a[10]; a+17
```

is well typed and evaluates to $((0, 17), 0)$, a nonexistent pointer. This leads us to define an *address typing* λ to be a finite function mapping *segment numbers* to data types. The notational conventions for address typings are like those for identifier typings.

We now modify our typing judgments to include an address typing:

$$\lambda; \gamma \vdash e : \rho$$

All of the rules given previously in Figures 1 and 2 need to be extended to include address typings, and we also add the new typing rules given in Figure 5. Furthermore, Figure 5 includes an updated version of rule (FUN) from Figure 2. In addition to including an address typing λ, the new rule replaces Close_{γ} with $\mathit{Close}_{\lambda;\gamma}$, which does not generalize type variables that are free in either λ or in γ.

To prove the type preservation theorem, we require a number of lemmas that establish some useful properties of the type system. We begin with a basic lemma that shows that our type system types closed values reasonably—it shows that any closed value of some type has the *form* that one would expect. It also shows that a closed expression of type τ *var* can have only two possible forms. (Note that $\emptyset$ here denotes an *empty* identifier typing.)

$$\text{(}\textsc{var}\text{)} \qquad \lambda;\gamma \vdash ((i,j),1) : \tau\ \mathit{var} \quad \text{if } \lambda(i) = \tau$$

$$\text{(}\textsc{ptr}\text{)} \qquad \lambda;\gamma \vdash ((i,j),0) : \tau\ \mathit{ptr} \quad \text{if } \lambda(i) = \tau$$

$$\text{(}\rightarrow\text{-}\textsc{intro}\text{)} \qquad \frac{\lambda;\gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau}{\lambda;\gamma \vdash \lambda x_1, \ldots, x_n.\, e : \tau_1 \times \cdots \times \tau_n \rightarrow \tau}$$

$$\text{(}\textsc{fun}\text{)} \qquad \frac{\begin{array}{l}\lambda;\gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau \\ \lambda;\gamma[x : \mathit{Close}_{\lambda;\gamma}(\tau_1 \times \cdots \times \tau_n \rightarrow \tau)] \vdash e' : \tau'\end{array}}{\lambda;\gamma \vdash x(x_1, \ldots, x_n)\ \ \{e\}\ \ e' : \tau'}$$

Fig. 5. New Rules for Typing Semantic Values

Lemma 2 (Correct Forms) *Suppose* $\lambda;\emptyset \vdash v : \tau$. *Then*

- *if* τ *is int, then* v *is an integer literal,*
- *if* τ *is unit, then* v *is* `unit`*,*
- *if* τ *is* τ' *ptr, then* v *is of the form* $((i,j),0)$*, and*
- *if* τ *is* $\tau_1 \times \cdots \times \tau_n \rightarrow \tau'$*, then* v *is of the form* $\lambda x_1, \ldots, x_n.e$*.*

And if $\lambda;\emptyset \vdash e : \tau\ var$*, then* e *is of the form* $((i,j),1)$ *or of the form* $*e'$*.*

PROOF. Immediate from inspection of the typing rules. (Note that the last part of the lemma assumes that array subscripting is syntactic sugar.) □

A consequence of the last part of this lemma is that if $\lambda;\emptyset \vdash e : \tau$ and e is not of the form $((i,j),1)$ or $*e'$, then the typing derivation cannot end with rule (R-VAL). So the typing rules, for the most part, remain syntax directed.

The fact that variables can have only two possible forms is also exploited in our evaluation rules, specifically within rules (REF) and (UPDATE) of Figure 3. In particular, we are able to define the semantics of `=` and `&` without defining an auxiliary relation for evaluation in "L-value" contexts; contrast our rules with those given in [3].

We continue with some basic lemmas showing that typings are preserved under substitutions and under extensions to the address and identifier typings:

Lemma 3 (Type Substitution) *If* $\lambda;\gamma \vdash e : \tau$*, then for any substitution* S*,* $S\lambda; S\gamma \vdash e : S\tau$*, and the latter typing has a derivation no higher than the former.*

PROOF. By induction on the structure of the derivation of $\lambda;\gamma \vdash e : \tau$. □

Lemma 4 (Superfluousness) *Suppose that* $\lambda;\gamma \vdash e : \tau$. *If* $i \notin \mathit{dom}(\lambda)$, *then* $\lambda[i : \tau'];\gamma \vdash e : \tau$, *and if* $x \notin \mathit{dom}(\gamma)$, *then* $\lambda;\gamma[x : \rho] \vdash e : \tau$.

PROOF. By induction on the height of the derivation of $\lambda;\gamma \vdash e : \tau$. The only way that adding an extra assumption can cause problems is by adding more free type variables to λ or γ, thereby preventing *Close* from generalizing such variables in (FUN) steps. If this happens, we must rename such variables in the original derivation before adding the extra assumption. By the Type Substitution Lemma, we can do this renaming and the height of the derivation is not increased. □

Lemma 5 (Substitution) *If* $\lambda;\gamma \vdash e : \rho$ *and* $\lambda;\gamma[x : \rho] \vdash e' : \tau$, *then* $\lambda;\gamma \vdash [e/x]e' : \tau$.

PROOF. Assume that the bound identifiers of e' are renamed as necessary to ensure that no identifier occurring in e occurs bound in e'. Then at every use of (IDENT) or (VAR-ID) on x in the derivation of $\lambda;\gamma[x : \rho] \vdash e' : \tau$, we can splice in the appropriate derivation for e. There may be extra assumptions around at that point, but by the Superfluousness Lemma, they do not cause problems. □

Lemma 6 (∀-intro) *If* $\lambda;\gamma \vdash e : \tau$ *and* $\alpha_1, \ldots, \alpha_n$ *do not occur free in* λ *or in* γ, *then* $\lambda;\gamma \vdash e : \forall \alpha_1, \ldots, \alpha_n \,.\, \tau$.

PROOF. This lemma is a simple corollary to the Type Substitution Lemma. Suppose that $\forall \bar{\alpha} \,.\, \tau \geq \tau'$. Then there exists a substitution $S = [\bar{\tau}/\bar{\alpha}]$ such that $S\tau = \tau'$. By the Type Substitution Lemma, $S\lambda; S\gamma \vdash e : S\tau$. Hence, since the $\bar{\alpha}$ are not free in λ or in γ, $\lambda;\gamma \vdash e : \tau'$. □

We now return to type preservation. Roughly speaking, we wish to show that if closed program e has type τ under address typing λ, and evaluates under memory μ to v, then v also has type τ. But since e can allocate addresses and these can occur in v, we cannot show that v has type τ under λ—we can only show that v has type τ under some address typing λ' that *extends* λ. (We denote "λ' extends λ" by $\lambda \subseteq \lambda'$.) Also, we need to assume that λ is *consistent* with μ—for example, if $\lambda(i) = \mathit{int}$, then μ needs to store integers in segment i. Precisely, we define $\mu : \lambda$ if

(i) $\mathit{dom}(\lambda) = \{i \mid (i, 0) \in \mathit{dom}(\mu)\}$, and

(ii) for all (i, j) such that $\mu((i, j))$ is a value, $\lambda \vdash \mu((i, j)) : \lambda(i)$.

Note that λ must give a type to uninitialized and dead addresses of μ, but the type can be anything. We can now prove the type preservation theorem:

Theorem 7 (Type Preservation) *If $\mu \vdash e \Rightarrow v, \mu'$, $\lambda; \emptyset \vdash e : \tau$, and $\mu : \lambda$, then there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda'; \emptyset \vdash v : \tau$.*

PROOF. By induction on the structure of the derivation of $\mu \vdash e \Rightarrow v, \mu'$. Here we just show the (BINDVAR) and (BINDFUN) cases; the remaining cases are similar.

(BINDVAR). The evaluation must end with

$$\frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ (i, 0) \notin \mathit{dom}(\mu_1) \\ \mu_1[(i, 0) := v_1] \vdash [((i, 0), 1)/x]e_2 \Rightarrow v_2, \mu_2 \end{array}}{\mu \vdash \texttt{var } x \texttt{ = } e_1\texttt{; } e_2 \Rightarrow v_2, \mu_2[(i, 0) := \textbf{dead}]}$$

while the typing must end with (LETVAR):

$$\frac{\begin{array}{l} \lambda; \emptyset \vdash e_1 : \tau_1 \\ \lambda; [x : \tau_1 \ \mathit{var}] \vdash e_2 : \tau_2 \end{array}}{\lambda; \emptyset \vdash \texttt{var } x \texttt{ = } e_1\texttt{; } e_2 : \tau_2}$$

and $\mu : \lambda$. By induction, there exists λ_1 such that $\lambda \subseteq \lambda_1$, $\mu_1 : \lambda_1$, and $\lambda_1; \emptyset \vdash v_1 : \tau_1$. Since $\mu_1 : \lambda_1$ and $(i, 0) \notin \mathit{dom}(\mu_1)$, also $i \notin \mathit{dom}(\lambda_1)$. So $\lambda_1 \subseteq \lambda_1[i : \tau_1]$. By rule (VAR),

$$\lambda_1[i : \tau_1]; \emptyset \vdash ((i, 0), 1) : \tau_1 \ \mathit{var}$$

and by Lemma 4,

$$\lambda_1[i : \tau_1]; [x : \tau_1 \ \mathit{var}] \vdash e_2 : \tau_2$$

So we can apply Lemma 5 to get

$$\lambda_1[i : \tau_1]; \emptyset \vdash [((i, 0), 1)/x]e_2 : \tau_2$$

Also, $\mu_1[(i, 0) := v_1] : \lambda_1[i : \tau_1]$. So by a second use of induction, there exists λ' such that $\lambda_1[i : \tau_1] \subseteq \lambda'$, $\mu_2 : \lambda'$, and $\lambda'; \emptyset \vdash v_2 : \tau_2$.

It only remains to show that $\mu_2[(i,0) := \textbf{dead}] : \lambda'$. But this follows immediately from $\mu_2 : \lambda'$.

Remark 8 What would go wrong if we simply removed the deallocated address $(i,0)$ from the domain of the final memory, rather than marking it **dead**? Well, with the current definition of $\mu : \lambda$, we would then be forced to remove i from the final address typing. But then $\mu_2 - i : \lambda' - i$ would fail, if there were any dangling pointers $((i,j),0)$ in the range of $\mu_2 - i$. If, instead, we allowed λ' to retain the typing for i, then the next time that $(i,0)$ were allocated we would have to *change* the typing for i, rather than *extend* the address typing.

(BINDFUN). The evaluation must end with

$$\frac{\mu \vdash [\lambda x_1, \ldots, x_n.\, e/x]e' \Rightarrow v, \mu'}{\mu \vdash x\,(x_1\,,\ldots,x_n)\;\;\{e\}\;\; e' \Rightarrow v, \mu'}$$

while the typing must end with (FUN):

$$\frac{\begin{array}{l}\lambda; [x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau \\ \lambda; [x : \mathit{Close}_{\lambda;\emptyset}(\tau_1 \times \cdots \times \tau_n \to \tau)] \vdash e' : \tau'\end{array}}{\lambda; \emptyset \vdash x\,(x_1\,,\ldots,x_n)\;\;\{e\}\;\; e' : \tau'}$$

and $\mu : \lambda$. By rule ($\to$-INTRO),

$$\lambda; \emptyset \vdash \lambda x_1, \ldots, x_n.\, e : \tau_1 \times \cdots \times \tau_n \to \tau$$

and so by Lemma 6,

$$\lambda; \emptyset \vdash \lambda x_1, \ldots, x_n.\, e : \mathit{Close}_{\lambda;\emptyset}(\tau_1 \times \cdots \times \tau_n \to \tau)$$

Therefore, by Lemma 5, $\lambda; \emptyset \vdash [\lambda x_1, \ldots, x_n.\, e/x]e' : \tau'$. So by induction, there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda' \vdash v : \tau'$. $\square$

5 Type Soundness

The type preservation property does not by itself ensure that a type system is sensible. For example, a type system that assigns *every* type to *every* expression trivially satisfies the type preservation property, even though such a type system is useless. The main limitation of type preservation is that it only applies to well-typed expressions that evaluate successfully. Really we would like to be able to say something about what happens when we *attempt* to evaluate an arbitrary well-typed expression.

One approach to strengthening type preservation (used by Gunter [6] and Harper [9], for example) is to augment the natural semantics with rules specifying that certain expressions evaluate to a special value, **TypeError**, which has no type. For example, an attempt to dereference a value other than a pointer would evaluate to **TypeError**. Then, by showing that type preservation holds for the augmented evaluation rules, we get that a well-typed expression cannot evaluate to **TypeError**. Hence any of the errors that lead to **TypeError** cannot occur in the evaluation of a well-typed expression. A drawback to this approach is the need to augment the natural semantics. But, more seriously, this approach does not give us as much information as we would like. It tells us that certain errors will not arise during the evaluation of well-typed expression, but it leaves open the possibility that there are *other* errors that we have neglected to check for in the augmented natural semantics.

Another approach is to use a different form of semantics than natural semantics. This is the approach advocated by Wright and Felleisen [25], who use a small-step structured operational semantics to prove type soundness for a number of extensions of ML. However, we find natural semantics to be much more *natural* and appealing than small-step structured operational semantics, particularly for languages with variables that have bounded lifetimes. (For example, in Ozgen's proposed small-step semantics for Polymorphic C [16], quite subtle mechanisms are employed to deallocate cells at the correct time.) Gunter and Remy [7] also propose an alternative to natural semantics, which they call *partial proof semantics*.

What we propose here is different. We argue that one can show a good type soundness theorem for a language, like Polymorphic C, defined using natural semantics. The trouble with natural semantics is that it defines only *complete* program executions, which are represented by derivation trees. But for a good type soundness theorem, we need a notion of an *attempted execution* of a program, which may of course fail in various ways. We argue, however, that a natural semantics gives rise in a natural way to a transition semantics, which we call a *natural transition semantics*, that provides the needed notion of an attempted program execution. [3]

The basic idea is that a program execution is a sequence of *partial derivation trees*, that may or may not eventually reach a complete derivation tree. In a partial derivation tree, some of the nodes may be labeled with *pending judgments*, which represent expressions that need to be evaluated in the program

[3] See [23] for a slightly different formulation of natural transition semantics; there, natural transition semantics is applied to a problem of computer security.

execution. A pending judgment is of the form $\mu \vdash e \Rightarrow ?$. (In contrast, we refer to ordinary judgments $\mu \vdash e \Rightarrow v, \mu'$ as *complete* judgments.)

Before we define partial derivation trees precisely, we need to make a few comments about the evaluation rules in a natural semantics. First, note that natural semantics rules are actually rule *schemas*, whose metavariables are instantiated in any use of the rule. Second, note that the hypotheses of each rule are either evaluation judgments $\mu \vdash e \Rightarrow v, \mu'$ or boolean conditions, such as the condition $a \in \mathit{dom}(\mu)$ in rule (CONTENTS). (Such boolean conditions are regarded as complete judgements.) Finally, note that in some hypotheses an evaluation judgment includes an implicit boolean condition. For example, the first hypothesis of rule (DEREF) is

$$\mu \vdash e \Rightarrow (a, 0), \mu'$$

This hypothesis is really an abbreviation for two hypotheses:

$$\mu \vdash e \Rightarrow v, \mu'$$

and

$$v \text{ is of the form } (a, 0)$$

Assume henceforth that we use the unabbreviated forms in derivation trees.

We want partial derivation trees to be limited to the trees that can arise in a systematic attempt to build a complete derivation tree; this constrains the form that such a tree can have. Precisely,

Definition 9 *A tree T whose nodes are labeled with (partial or complete) judgments is a* partial derivation tree *if it satisfies the following two conditions:*

(i) If a node in T is labeled with a complete judgment J, then the subtree rooted at that node is a complete derivation tree for J.

(ii) If a node in T is labeled with a pending judgment $\mu \vdash e \Rightarrow ?$ and the node has k children, where $k > 0$, then there is an instance of an evaluation rule that has the form

$$\frac{J_1 \; J_2 \; \ldots \; J_n}{\mu \vdash e \Rightarrow v, \mu'}$$

where $n \geq k$, and the labels on the children are $J_1, J_2, \ldots, J_k$, respectively, with possibly one exception: if J_k is $\mu_k \vdash e_k \Rightarrow v_k, \mu'_k$, then the kth child may alternatively be labeled with the pending judgment $\mu_k \vdash e_k \Rightarrow ?$.

One may readily see that a partial derivation tree can have at most one pending judgment on each level, which must be the rightmost node of the level, and whose parent must also be a pending judgment.

Next we define transitions, based on the rules of the natural semantics, that describe how one partial derivation tree can be transformed into another. Suppose that there is an instance of an evaluation rule that has the form

$$\frac{J_1\ J_2\ \ldots\ J_n}{\mu \vdash e \Rightarrow v, \mu'}$$

where each hypothesis J_i is either an evaluation judgment $\mu_i \vdash e_i \Rightarrow v_i, \mu'_i$ or else a boolean condition B_i.

The transformations resulting from this rule are defined as follows:

> Suppose that a partial derivation tree T contains a node N labeled with the pending judgment $\mu \vdash e \Rightarrow ?$ and that the children of N are labeled with the complete judgments $J_1, J_2, \ldots, J_k$ where $0 \leq k$.
>
> - Suppose $k < n$. Then if J_{k+1} is of the form $\mu_{k+1} \vdash e_{k+1} \Rightarrow v_{k+1}, \mu'_{k+1}$, we can transform T by adding another child to N, labeled with the pending judgment $\mu_{k+1} \vdash e_{k+1} \Rightarrow ?$. And if J_{k+1} is a boolean condition B_{k+1} that is true, we can transform T by adding another child to N, labeled with B_{k+1}.
> - Now suppose $k = n$. Then we can transform T by replacing the label on N with the complete judgement $\mu \vdash e \Rightarrow v, \mu'$.

We write $T \longrightarrow T'$ if partial derivation tree T can be transformed in one step to T'. As usual, $\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$.

Remark 10 We remark that, in the case of Polymorphic C, the transformation relation thus defined is almost deterministic. In particular, although there are two evaluation rules for `if` (e_1) $\{e_2\}$ `else` $\{e_3\}$ and `while` (e_1) $\{e_2\}$, there is no ambiguity, since we need not choose which rule is being applied until after the guard e_1 has been evaluated. The only nondeterminism in the transformation relation is in rules (BINDVAR) and (BINDARR). The second hypothesis of both rules is $(i, 0) \notin dom(\mu_1)$, and here metavariable i is not bound deterministically. But, of course, this nondeterministic choice of an address for a newly-allocated variable or array is of no importance. □

A key property of $\longrightarrow$ is that it always transforms a partial derivation tree into another partial derivation tree:

Lemma 11 *If T is a partial derivation tree and $T \longrightarrow T'$, then T' is also a partial derivation tree.*

PROOF. Straightforward. $\square$

The transformation rules give us the desired notion of program execution: to execute e in memory μ, we start with the tree T_0 which consists of a single root node labeled with the pending judgment $\mu \vdash e \Rightarrow ?$, and then we apply the transformations, generating a sequence of partial derivation trees:

$$T_0 \longrightarrow T_1 \longrightarrow T_2 \longrightarrow T_3 \longrightarrow \cdots$$

More precisely, we define an *execution* of program e in memory μ to be a possibly infinite sequence of partial derivation trees $T_0, T_1, T_2, \ldots$ such that

- T_0 is a one-node tree labeled with $\mu \vdash e \Rightarrow ?$,
- for all $i \geq 0$, $T_i \longrightarrow T_{i+1}$ (unless T_i is the last tree in the sequence), and
- if the sequence has a last tree T_n, then there is no tree T such that $T_n \longrightarrow T$.

Note that there are three possible outcomes to an execution:

(i) The sequence ends with a complete derivation tree. This is a *successful* execution.
(ii) The sequence is infinite. This is a *nonterminating* execution.
(iii) The sequence ends with a tree T_n that contains a pending judgment but has no successor. This is an *aborted* execution.

Our Type Soundness theorem will show that, for well-typed programs, aborted execution can arise only from one of a specific set of errors.

But first, we argue that our notion of execution is correct. Let us write $[J]$ to denote the one-node tree labeled with J. The soundness of our notion of execution is given by the following lemma.

Lemma 12 *If $[\mu \vdash e \Rightarrow ?] \longrightarrow^* T$, where T contains no pending judgments, then T is a complete derivation tree for a judgment of the form $\mu \vdash e \Rightarrow v, \mu'$.*

PROOF. By Lemma 11, T is a partial derivation tree. So, since T contains no pending judgments, T is a complete derivation tree for the judgment that labels its root. And this judgment must be of the form $\mu \vdash e \Rightarrow v, \mu'$, because the initial tree has a root labeled with $\mu \vdash e \Rightarrow ?$ and (as can be seen by inspecting the definition of $\longrightarrow$) the only transformation that changes the label on a node changes a label of the form $\mu \vdash e \Rightarrow ?$ to a label of the form $\mu \vdash e \Rightarrow v, \mu'$. $\square$

Next we show that our notion of execution is complete:

Lemma 13 *If $\mu \vdash e \Rightarrow v, \mu'$ and T is a complete derivation tree for $\mu \vdash e \Rightarrow v, \mu'$, then $[\mu \vdash e \Rightarrow ?] \longrightarrow^* T$.*

PROOF. By induction on the structure of the derivation of $\mu \vdash e \Rightarrow v, \mu'$. □

Remark 14 This lemma shows that if $\mu \vdash e \Rightarrow v, \mu'$, then there is a successful execution of e in μ. But it does not show that *every* execution of e in μ is successful. With an arbitrary natural semantics, this need not be so. For example, in a language with a nondeterministic choice operator, some executions of e in μ may be successful, others may be nonterminating, and others may abort. But in Polymorphic C, since $\longrightarrow$ is essentially deterministic, a stronger result should hold. □

Now that we have a notion of program execution, we again turn to Polymorphic C and consider what we can say about the executions of well-typed Polymorphic C programs.

Definition 15 *A pending judgment $\mu \vdash e \Rightarrow ?$ is* well typed *iff there exist an address typing λ and a type τ such that $\mu : \lambda$ and $\lambda; \emptyset \vdash e : \tau$. Also, a partial derivation tree T is* well typed *iff every pending judgment in it is well typed.*

Roughly speaking, the combination of the Type Preservation theorem and the Correct Forms lemma (Lemma 2) allows us to characterize the forms of expressions that will be encountered during the execution of a well-typed program. This allows us to characterize what can go wrong during the execution. Here is the key type soundness result:

Theorem 16 (Progress) *Let T be a well-typed partial derivation tree that contains at least one pending judgment. If $T \longrightarrow T'$, then T' is well typed. Furthermore, there exists T' such that $T \longrightarrow T'$, unless T contains one of the following errors:*

E1. A read or write to a dead address.
E2. A read or write to an address with an invalid offset.
E3. A read of an uninitialized address.
E4. A declaration of an array of size 0 or less.

PROOF. Let N be the uppermost node in T that is labeled with a pending judgment, say $\mu \vdash e \Rightarrow ?$. Then any transformation on T must occur at this node. We just consider all possible forms of expression e. Here we just give the case e_1=e_2; the other cases are quite similar.

Since T is well typed, the pending judgment $\mu \vdash e_1\texttt{=}e_2 \Rightarrow ?$ is well typed, and so there exist λ and τ such that $\mu : \lambda$ and $\lambda; \emptyset \vdash e_1\texttt{=}e_2 : \tau$. The latter typing must be by (ASSIGN):

$$\frac{\begin{array}{l}\lambda; \emptyset \vdash e_1 : \tau\ \mathit{var} \\ \lambda; \emptyset \vdash e_2 : \tau\end{array}}{\lambda; \emptyset \vdash e_1\texttt{=}e_2 : \tau}$$

By the Correct Forms lemma, e_1 must be of the form $((i,j),1)$ or else of the form $*e_1'$. So, simplifying notation a bit, the pending judgment that labels N has the form $\mu \vdash (a,1)\texttt{=}e \Rightarrow ?$ or $\mu \vdash *e_1\texttt{=}e_2 \Rightarrow ?$. We consider these two cases in turn.

If the label of N is $\mu \vdash (a,1)\texttt{=}e \Rightarrow ?$, where $\mu : \lambda$ and $\lambda; \emptyset \vdash (a,1)\texttt{=}e : \tau$, then the typing must end with (ASSIGN):

$$\frac{\begin{array}{l}\lambda; \emptyset \vdash (a,1) : \tau\ \mathit{var} \\ \lambda; \emptyset \vdash e : \tau\end{array}}{\lambda; \emptyset \vdash (a,1)\texttt{=}e : \tau}$$

So by (VAR), a is of the form (i,j), where $\lambda(i) = \tau$.

Now, if N has no children, then (using rule (UPDATE)), we can transform T by adding to N a new child, labeled with the pending judgment $\mu \vdash e \Rightarrow ?$. Furthermore, this is the only possible transformation, and since $\lambda; \emptyset \vdash e : \tau$, this new pending judgment is well typed.

If N has exactly one child, then by condition *(ii)* of the definition of partial derivation tree and the fact that N is the uppermost node labeled with a pending judgment, it must be that the child of N is labeled with a judgment of the form $\mu \vdash e \Rightarrow v, \mu'$. In this case, we may transform T by adding a new child to N labeled with the boolean condition

$$a \in \mathit{dom}(\mu') \text{ and } \mu'(a) \neq \mathbf{dead}$$

provided that this condition is true.

Now, by the Type Preservation theorem, there exists λ' such that $\lambda \subseteq \lambda'$, $\mu' : \lambda'$, and $\lambda'; \emptyset \vdash v : \tau$. Hence $\lambda'(i) = \tau$, and so $(i,0) \in \mathit{dom}(\mu')$. So if $(i,j) \notin \mathit{dom}(\mu')$, then T contains error *E2*, a write to an address with an invalid offset j. And if $\mu'((i,j)) = \mathbf{dead}$, then T contains error *E1*, a write to a dead address. Hence we can transform T unless it contains error *E2* or *E1*.

Finally, if N has two children, then they must be labeled with the hypotheses of rule (UPDATE), and so we can transform T by replacing the label of N with $\mu \vdash (a,1)\texttt{=}e \Rightarrow v, \mu'[a := v]$.

If the label of N is $\mu \vdash *e_1\texttt{=}e_2 \Rightarrow ?$, where $\mu : \lambda$ and $\lambda; \emptyset \vdash *e_1\texttt{=}e_2 : \tau$, then the typing must end with (L-VAL) followed by (ASSIGN):

$$\dfrac{\begin{array}{l}\dfrac{\lambda; \emptyset \vdash e_1 : \tau\ ptr}{\lambda; \emptyset \vdash *e_1 : \tau\ var} \\ \lambda; \emptyset \vdash e_2 : \tau\end{array}}{\lambda; \emptyset \vdash *e_1\texttt{=}e_2 : \tau}$$

Now, if N has no children, then the only applicable transformation (using rule (UPDATE)) is to add to N a new child, labeled with the pending judgment $\mu \vdash e_1 \Rightarrow ?$. Since $\lambda; \emptyset \vdash e_1 : \tau\ ptr$, this new pending judgment is well typed.

If N has exactly one child, then by condition *(ii)* of the definition of partial derivation tree and the fact that N is the uppermost node labeled with a pending judgment, it must be that the child of N is labeled with a judgment of the form $\mu \vdash e_1 \Rightarrow v_1, \mu_1$.

By the Type Preservation theorem, there exists λ_1 such that $\lambda \subseteq \lambda_1$, $\mu_1 : \lambda_1$, and $\lambda_1; \emptyset \vdash v_1 : \tau\ ptr$. So by the Correct Form lemma, v_1 is of the form $((i,j),0)$. Hence, we may transform T by adding a new child to N labeled with the boolean condition

$$v_1 \text{ is of the form } (a, 0),$$

since this is guaranteed to be true. Also, by (PTR), $\lambda_1(i) = \tau$.

If N has two children, then we can transform T by adding a new child labeled with the pending judgment $\mu_1 \vdash e_2 \Rightarrow ?$. By the Superfluousness Lemma, $\lambda_1; \emptyset \vdash e_2 : \tau$, so this pending judgment is well typed.

If N has three children, then the third child of N must be labeled with a judgment of the form $\mu_1 \vdash e_2 \Rightarrow v, \mu_2$. In this case, we may transform T by adding a new child to N labeled with the boolean condition

$$a \in \mathit{dom}(\mu_2) \text{ and } \mu_2(a) \neq \mathbf{dead}$$

provided that this condition is true.

As before, by the Type Preservation theorem, there exists λ' such that $\lambda_1 \subseteq \lambda'$,

$\mu_2 : \lambda'$, and $\lambda'; \emptyset \vdash v : \tau$. Hence $\lambda'(i) = \tau$, and so $(i, 0) \in dom(\mu_2)$. So if $(i, j) \notin dom(\mu_2)$, then T contains error *E2*, a write to an address with an invalid offset j. And if $\mu_2((i, j)) = \textbf{dead}$, then T contains error *E1*, a write to a dead address. Hence we can transform T unless it contains error *E2* or *E1*.

Finally, if N has four children, then they must be labeled with the hypotheses of rule (UPDATE), and so we can transform T by replacing the label of N with $\mu \vdash *e_1 = e_2 \Rightarrow v, \mu_2[a := v]$. □

The Progress theorem gives our Type Soundness result as a simple corollary:

Corollary 17 (Type Soundness) *If $\lambda; \emptyset \vdash e : \tau$ and $\mu : \lambda$, then any execution of e in μ either*

(i) succeeds,
(ii) does not terminate, or
(iii) aborts due to one of the errors E1, E2, E3, or E4.

PROOF. Let $T_0 \longrightarrow T_1 \longrightarrow T_2 \longrightarrow \cdots$ be an execution of e in μ. Then $T_0 = [\mu \vdash e \Rightarrow ?]$, which is well typed by assumption. So, by the Progress theorem, every T_i is well typed, and furthermore, if T_i contains a pending judgment, then it has a successor unless it contains one of the errors *E1*, *E2*, *E3*, or *E4*. So, if the execution is finite, it either ends with a complete derivation tree or with a tree containing one of the errors *E1*, *E2*, *E3*, or *E4*. □

6 Discussion

One of the most desirable properties of a programming language implementation is that it guarantee the *safe execution* of programs. This means that a program's execution is always faithful to the language's semantics, even if the program is erroneous. C is, of course, a notoriously *unsafe* language: in typical implementations, pointer errors can cause a running C program to overwrite its runtime stack, resulting in arbitrarily bizarre behavior. Sometimes this results in a "Segmentation fault—core dumped" message (though this may occur far after the original error); worse, at other times the program appears to run successfully, even though the results are entirely invalid.

Three techniques can be used to provide safe execution:

(i) The language can be designed so that some errors are impossible. For example, a language can define default initializations for variables, thereby preventing uninitialized variable errors.

(ii) The language can perform compile-time checks, such as type checks, to guard against other errors.
(iii) Finally, runtime checks can be used to catch other errors.

In the case of Polymorphic C, the Type Soundness theorem (Corollary 17) specifies exactly what runtime checks are needed to guarantee safe execution. The trouble is, except for error *E4* (declaring an array of size 0 or less), typical C implementations do *not* make these checks. What would we expect, then, of implementations of Polymorphic C? Well, it is actually not too difficult to check for error *E2* (reading or writing an address with an invalid offset)—for each pointer, we must maintain at runtime the range of permissible offsets. And error *E3* (reading an uninitialized address) can also be checked fairly efficiently, by initializing array cells with a special **uninit** value. That leaves only error *E1* (reading or writing a dead address). This, of course, is very difficult to check efficiently. In our natural semantics, we make this check possible by never reusing any cells!

Hence we reach a point of trade-offs. We can directly implement our natural semantics, getting a safe but inefficient "debugging" implementation of Polymorphic C. Or we can follow usual C practice and build a stack-based implementation that leaves errors *E1* (and perhaps *E2* and *E3* as well) unchecked, achieving efficiency at the expense of safety.[4] In this case, the Type Soundness theorem at least tells us what *kinds* of errors we need to look for in debugging our programs. As a final alternative, we can change the semantics of Polymorphic C by giving cells unbounded lifetimes (thereby necessitating garbage collection), as was done in the design of Java [1].

7 Conclusion

Advanced polymorphic type systems have come to play a central role in the world of functional programming, but so far have had little impact on traditional imperative programming. We assert that an ML-style polymorphic type system can be applied fruitfully to a "real-world" language like C, bringing to it both the expressiveness of polymorphism as well as a rigorous characterization of the behavior of well-typed programs.

Future work on Polymorphic C includes the development of efficient implementations of polymorphism (perhaps using the work of [13,18,10]) and the extension of the language to include other features of C, especially structures.

[4] More precisely, allocating variables and arrays on a stack in Polymorphic C (or in any language with & or that unifies arrays and pointers) causes the type preservation property to fail.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, New York, 1982. ACM.

[3] Pascal Fradet, Ronan Gaugne, and Daniel Le Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In *Proceedings of the 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 125–140, Berlin, April 1996. Springer-Verlag.

[4] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[5] John Greiner. Standard ML weak polymorphism can be sound. Technical Report CMU-CS-93-160, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., May 1993.

[6] Carl Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[7] Carl Gunter and Didier Rémy. A proof-theoretic assessment of runtime type errors. Technical Report 11261-921230-43TM, AT&T Bell Laboratories, 1993.

[8] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, August 1994.

[9] Robert Harper. A note on "A simplified account of polymorphic references". *Information Processing Letters*, 57:15–16, January 1996.

[10] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, New York, 1995. ACM.

[11] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML/NJ weak polymorphism and imperative constructs. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science*, pages 15–25, New York, 1993. IEEE.

[12] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[13] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 177–188, New York, 1992. ACM.

[14] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 291–302, New York, 1991. ACM.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[16] Mustafa Ozgen. A type inference algorithm and transition semantics for Polymorphic C. Master's thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, September 1996.

[17] John C. Reynolds. The essence of ALGOL. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland Publishing Company, 1981.

[18] Zhong Shao and Andrew Appel. A typed-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, 1995.

[19] Geoffrey Smith and Dennis Volpano. Polymorphic typing of variables and references. *ACM Transactions on Programming Languages and Systems*, 18(3):254–267, May 1996.

[20] Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *Proceedings of the 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 341–355, Berlin, April 1996. Springer-Verlag.

[21] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[22] Dennis Volpano and Geoffrey Smith. A type soundness proof for variables in LCF ML. *Information Processing Letters*, 56:141–146, 1995.

[23] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168. IEEE, June 1997.

[24] Andrew Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[25] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.